

AX.25 Version 2.2

John Langner, WB2OSZ

December 2025

If you had to choose between using a mobile phone from 1984 and one from 1998, the decision would be obvious. In just fourteen years, mobile technology evolved from bulky, single-purpose bricks into compact, reliable, feature-rich devices. What's easy to forget is that communication protocols evolve in exactly the same way — and Amateur Packet Radio is no exception.

Most operators know that AX.25 is the protocol that made amateur packet radio possible. What fewer realize is that **there isn't just one AX.25 standard**. In fact, there are two major revisions, separated by nearly a decade and a half of real-world experience, experimentation, and lessons learned.

Packet radio exploded in popularity in the mid-1980s. New TNC manufacturers appeared almost overnight, and the hobby grew rapidly because everyone agreed to speak the same digital “language”:

- **AX.25 Version 2.0 — October 1984**

But technology didn't stand still. After more than ten years of operational use, network experimentation, and feedback from thousands of packet operators, the standard was revisited and improved. The result was a significantly more capable and efficient protocol:

- **AX.25 Version 2.2 — July 1998** *(now more than 25 years old as I write this)*

Ironically, while the 1998 revision brought meaningful improvements, existing implementations rarely adopted it. Most manufacturers disappeared, and the few that remain don't mention AX.25 2.2 support at all.

So, what changed? And why does AX.25 Version 2.2 offer such clear advantages over its 1984 predecessor?

Let's take a closer look at what makes Version 2.2 the better protocol — and why you should be using it.

1 APRS vs. Connected mode Packet

The AX.25 protocol is used mainly in two ways:

APRS	Connected mode
One to many.	One to one.
Usually no particular destination.	Establish a link with a specific station.
Send and forget independent (unnumbered) information frames.	The link protocol assigns sequence numbers to the information frames.
Don't know if frames were received by anyone.	The receiving end replies with acknowledgements that the frames were received. Lost frames are resent automatically and delivered in the proper order.

The primary advantage of AX.25 Version 2.2 is its significantly improved efficiency in connected mode. This comes from two key enhancements:

- a larger, more flexible window size
- a far more robust error-recovery mechanism

Let's explore these improvements in more detail.

2 MAXFRAME

In connected mode, each transmitted frame is tagged with a sequence number, acknowledged by the receiving station, and retransmitted if the acknowledgement doesn't arrive. The number of outstanding frames a station may send before it must pause for an acknowledgement is called the **window size**, commonly configured in a TNC through the **MAXFRAME** parameter.

If Station A had 4 information frames for Station B, the communication might go like this.

Station A	Transmit Direction	Transmit Direction	Station B
Information frame 0	→	←	Got it; Ready for 1
Information frame 1	→	←	Got it; Ready for 2
Information frame 2	→	←	Got it; Ready for 3
Information frame 3	→	←	Got it; Ready for 4

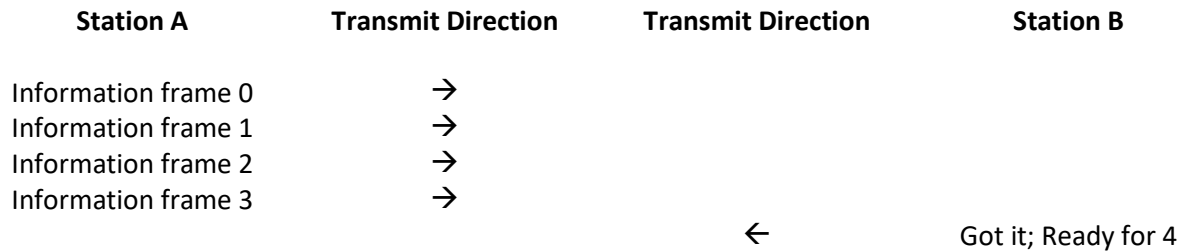
This approach is highly inefficient — every other transmission is simply an acknowledgement (ACK) for a single Information frame. The **MAXFRAME** parameter, also known as the **window size**, defines how many Information frames can be sent before the sender must pause and wait for an acknowledgement. In the example above, MAXFRAME is set to 1.

If we increase MAXFRAME to 2, the exchange would look like this:

Station A	Transmit Direction	Transmit Direction	Station B
Information frame 0	→		
Information frame 1	→		
		←	Got it; Ready for 2
Information frame 2	→		
Information frame 3	→		
		←	Got it; Ready for 4

With a window size of two, a single ACK can confirm both frames, reducing acknowledgements from half of all traffic to just one-third. Because two Information frames can be sent back-to-back, the random wait time and TXDELAY between transmissions are eliminated. It's an immediate efficiency gain.

And why stop at two? With **MAXFRAME = 4**, four frames can be sent in one continuous burst, and the receiving station can acknowledge all of them with a single response.



We'll look at the tradeoffs — and the solution — a bit later, but the key point for now is simple: **larger windows are more efficient**. A later section will explain what happens when frames are lost or corrupted.

In AX.25 Version 2.0, the window size is limited to **7**, a consequence of its 3-bit sequence numbering. Most TNCs default to a value of **4**.

So why not just make the window as large as possible? The answer becomes clear once we examine how error handling works, which we'll cover shortly.

3 EMAXFRAME

AX.25 Version 2.2 introduces **extended sequence numbers**, expanding them to 7 bits. This allows a sender to transmit **up to 63 frames** before pausing for an acknowledgement (ACK). Although 7 bits could represent 127 values, only half the range is usable once we incorporate **Selective Reject**, which we'll discuss later. In practice, the default window size is **32**.

window size	v 2.0 MAXFRAME	v 2.2 EMAXFRAME
Minimum	1	1
Default	4	32
Maximum	7	63

Remember: a larger number is a win-win:

- Only a single random delay and TXDELAY for many frames sent in a single transmission.
- Single acknowledgement for many Information frames.

Until, we consider what happens when a transmission error occurs.

4 When a frame gets lost

Up to this point, we've assumed the ideal situation where every frame arrives without errors. In the real world, of course, some frames will be lost or corrupted. So what happens then?

Imagine Station A sends six Information frames in a single transmission.

Station A	Transmit Direction	Transmit Direction	Station B	Given to user
Information frame 0	→			I frame 0
Information frame 1	→ (gets clobbered)		(not received)	
Information frame 2	→		(Discarded)	
Information frame 3	→		(Discarded)	
Information frame 4	→		(Discarded)	
Information frame 5	→		(Discarded)	
		←	Got it; Ready for 1	
Information frame 1	→			I frame 1
Information frame 2	→			I frame 2
Information frame 3	→			I frame 3
Information frame 4	→			I frame 4
Information frame 5	→			I frame 5
		←	Got it; Ready for 6	

AX.25 Version 2.0 does not handle error recovery particularly well. Imagine the following scenario: the first Information frame arrives correctly, has the expected sequence number, and is delivered to the operator or application.

- Frame 1 is lost and never reaches the destination.
- Frames 2, 3, 4, and 5 arrive intact, but because Frame 1 is missing, they are considered “out of order” and are discarded.
- When the transmission ends, the receiver requests a retransmission beginning with the missing frame.
- Frames 1, 2, 3, 4, and 5 are all sent again.

You're probably thinking, “This is wasteful — frames 2 through 5 were fine. Why not keep them and just ask for Frame 1?” Hold that thought; the next section explains why Version 2.0 can't do that — and how Version 2.2 fixes it.

The AX.25 v2.0 specification was created at a time when memory was both limited and expensive. Buffering out-of-sequence frames simply wasn't practical for many small,

resource-constrained TNC designs. The result is an important limitation: if the window size is large, losing even a single frame may force the sender to retransmit many frames that were already received correctly.

This leads to a fundamental tradeoff in AX.25 v2.0:

- Larger window sizes improve efficiency by reducing overhead and increasing throughput.
- But in environments where frames are occasionally lost, larger windows make retransmissions far more wasteful.

Fortunately, the 1998 AX.25 v2.2 update introduced a solution — though, to date, very few implementations have taken advantage of it.

5 Selective Reject

Earlier we noted that a very large window (MAXFRAME) can backfire, forcing the sender to retransmit many frames that were already received correctly. AX.25 Version 2.2 addresses this with a new feature called **Selective Reject (SREJ)**. The name is a bit counterintuitive — “selective resend” would describe the behavior more accurately.

In the following example, Station A once again sends seven Information frames in a single transmission.

Station A	Transmit Direction	Transmit Direction	Station B	Given to user
Information frame 0	→			I frame 0
Information frame 1	→			I frame 1
Information frame 2	→ (gets clobbered)		(not received)	
Information frame 3	→		(save 3 in memory)	
Information frame 4	→		(save 4 in memory)	
Information frame 5	→ (gets clobbered)		(not received)	
Information frame 6	→		(save 6 in memory)	
		←	Resend 2 & 5	
Information frame 2	→			I frame 2
				I frame 3
				I frame 4
Information frame 5	→			I frame 5
				I frame 6
		←	Got it; Ready for 7	

This approach is far more efficient.

- Frames 3, 4, and 6 arrive out of order, so the receiver temporarily stores them.
- It detects that frames 2 and 5 are missing and requests retransmission of only those specific frames.
- The sender resends just the missing frames.
- The receiver then reassembles everything in the correct order and delivers it cleanly to the user application.

With Selective Reject, large window sizes no longer carry a retransmission penalty.

Only the two end nodes need to support Selective Reject for it to function. Digipeaters simply forward frames based on the digipeater path and do not care whether the endpoints are using AX.25 v2.0 or v2.2.

6 AX.25 v2.0 & v2.2 interoperability

Throughout this document, we've referred repeatedly to the AX.25 v2.2 revision. Unfortunately, most implementations created before 1998 were never updated to include the improvements introduced in that standard. This naturally raises a question: **can v2.0 and v2.2 systems communicate with each other?**

Yes - they can.

What happens when an older implementation requests a connection?

Originating Station - V2.0	Transmission direction	Transmission direction	Answering Station - Either version	Comment
SABM	→			Request v2.0 connection
		←	UA	Accepted. We will use v2.0

What happens when a newer implementation connects to newer one?

Originating Station - V2.2	Transmission direction	Transmission direction	Answering Station - V2.2	Comment
SABME	→			Request v2.2 connection (note SABME vs. SABM)
		←	UA	Accepted. We will use v2.2

And, now, what you have been waiting for....

What happens when a newer implementation connects to an older one?

Originating Station - V2.2	Transmission direction	Transmission direction	Answering Station - V2.0	Comment
SABME	→			Request v2.2 connection
		←	FRMR	Invalid command. (I have no idea what you are talking about.)
SABM	→			Request v2.0 connection
		←	UA	Accepted. We will use v2.0

The v2.0 protocol specification clearly states:

2.3.4.3.3 Frame Reject (FRMR) Response

2.3.4.3.3.1

The FRMR response frame is sent to report that the receiver of a frame cannot successfully process that frame and that the error condition is not correctable by sending the offending frame again. Typically this condition will appear when a frame without an FCS error has been received with one of the following conditions:

1. *The reception of an invalid or not implemented command or response frame.*
2. *[others ...]*

An invalid or not implemented command or response is defined as a frame with a control field that is unknown to the receiver of this frame.

SABME was not defined in the AX.25 v2.0 protocol. When an older implementation receives this unfamiliar command, it responds with an FRMR frame. AX.25 v2.2, however, never sends FRMR under any circumstances. So when a v2.2 station receives an FRMR, it recognizes that it is communicating with a v2.0 peer and automatically falls back to the older protocol.

The two versions are fully interoperable. When both stations support v2.2, they take advantage of its enhanced features. When a v2.0 and v2.2 station communicate, they simply operate using the older version.

7 Summary

AX.25 Version 2.2 offers significant improvements over the 1984 v2.0 standard. So why continue relying on an implementation rooted in technology from four decades ago?

This brief introduction only touches on the highlights. For a deeper exploration, see:

[AX.25 Throughput: Why is 9600 bps Packet Radio only twice as fast as 1200?](#)

AX.25 v2.2 brings meaningful, long-overdue improvements to a protocol many of us still rely on every day. Its extended sequence numbers, selective retransmission, and more efficient connected-mode operation make it a clear upgrade over the 1984 v2.0 standard that most implementations continue to use.

If your station, TNC, or software stack is still anchored to v2.0, it may be time to consider what you're missing. The enhancements in v2.2 aren't theoretical; they translate directly into smoother links, higher throughput, and far less wasted airtime.

This introduction only scratches the surface. For a deeper dive into the protocol, its history, and its modern applications, you may want to explore the full AX.25 v2.2 specification, implementation notes from TNC developers, and community resources that examine the protocol in real-world use.

If you're still relying on AX.25 v2.0, now is the perfect moment to rethink that choice. The improvements in v2.2 translate directly into faster links, fewer retries, and far more efficient use of the shared radio channel. Whether you're developing software, maintaining a TNC, or simply aiming to get the most out of your station, adopting (or advocating for) v2.2 support is one of the most impactful upgrades you can make.

The protocol has evolved. Our tools and implementations should too.